**DATA STRUCTURES AND ALGORITHMS(HANDS\_ON)**

Exercise 2: E-commerce Platform Search Function

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

SOLUTION:

Big O notation is a mathematical concept used in computer science to describe the efficiency of an algorithm. It provides an upper bound on the growth rate of an algorithm's time or space complexity as the input size increases.

| **Big O** | **Name** | **Description** | **Example Algorithm** |
| --- | --- | --- | --- |
| O(1) | Constant time | Executes in the same time regardless of input size. | Accessing array element |
| O(log n) | Logarithmic time | Reduces input size each step | Binary search |
| O(n) | Linear time | Time grows directly with input size | Linear search,traversal |
| O(n logn) | Log-linear time | Efficient sorting | Merge sort,quick sort |
| O(n^2) | Quadratic time | Time grows with square of input | Bubble sort, nested loops |
| O(2^n) | Exponential time | Doubles with each input increase | Recursive Fibonacci |
| O(n!) | Factorial time | Extremely slow for large inputs | Brute-force permutations |

**b. Describe the best, average, and worst-case scenarios for search operations.**

For search operations, we are typically looking for an item in a collection of n items.

* **Best Case:**
  + **Definition:** The scenario where the algorithm performs the minimum number of operations.
  + **Example (Linear Search):** The target item is the very first element checked. Complexity: O(1).
  + **Example (Binary Search):** The target item is the middle element of the sorted collection, found on the first check. Complexity: O(1).
* **Average Case:**
  + **Definition:** The expected performance of the algorithm over all possible inputs, assuming a uniform distribution of inputs or a typical input.
  + **Example (Linear Search):** The target item is found somewhere in the middle of the collection, or it's not present and we have to check about half the elements on average (though formally still O(n) as we're interested in the upper bound of growth). For a successful search, on average, n/2 comparisons.
  + **Example (Binary Search):** The target item is found after a few divisions of the search space. Complexity: O(log n).
* **Worst Case:**
  + **Definition:** The scenario where the algorithm performs the maximum number of operations. Big O notation typically focuses on this.
  + **Example (Linear Search):** The target item is the last element checked, or the item is not present in the collection at all, requiring a full scan. Complexity: O(n).
  + **Example (Binary Search):** The target item is found at the very end of the search (after maximum divisions), or the item is not present, requiring the search space to be narrowed down completely. Complexity: O(log n).

CODE:

>**Product.cs**

using System;

class Product

{

    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public string Category { get; set; }

    public Product(int id, string name, string category)

    {

        ProductId = id;

        ProductName = name;

        Category = category;

    }

    public override string ToString()

    {

        return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";

    }

}

>**EcommerceSearch.cs**

class ECommerceSearch

{

public static Product LinearSearch(Product[] products, string name)

{

foreach (var product in products)

{

if (product.ProductName.Equals(name, StringComparison.OrdinalIgnoreCase))

{

return product;

}

}

return null;

}

public static Product BinarySearch(Product[] products, string name)

{

int left = 0, right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

int cmp = string.Compare(products[mid].ProductName, name, StringComparison.OrdinalIgnoreCase);

if (cmp == 0) return products[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

static void PrintTable(Product[] products)

{

Console.WriteLine("\n-------------------- Product List --------------------");

Console.WriteLine($"{"Product ID",-10} {"Product Name",-20} {"Category",-15}");

Console.WriteLine("------------------------------------------------------");

foreach (var p in products)

{

Console.WriteLine(p.ToString());

}

Console.WriteLine("------------------------------------------------------\n");

}

static void Main()

{

Console.Write("Enter number of products: ");

int n = int.Parse(Console.ReadLine());

Product[] unsortedProducts = new Product[n];

for (int i = 0; i < n; i++)

{

Console.WriteLine($"\nEnter details for Product {i + 1}:");

Console.Write("Product ID: ");

int id = int.Parse(Console.ReadLine());

Console.Write("Product Name: ");

string name = Console.ReadLine();

Console.Write("Category: ");

string category = Console.ReadLine();

unsortedProducts[i] = new Product(id, name, category);

}

PrintTable(unsortedProducts);

Product[] sortedProducts = (Product[])unsortedProducts.Clone();

Array.Sort(sortedProducts, (a, b) => a.ProductName.CompareTo(b.ProductName));

Console.Write("Enter product name to search: ");

string searchName = Console.ReadLine();

var resultLinear = LinearSearch(unsortedProducts, searchName);

Console.WriteLine("\n--- Linear Search Result ---");

Console.WriteLine(resultLinear != null ? resultLinear.ToString() : "Product not found.");

var resultBinary = BinarySearch(sortedProducts, searchName);

Console.WriteLine("\n--- Binary Search Result ---");

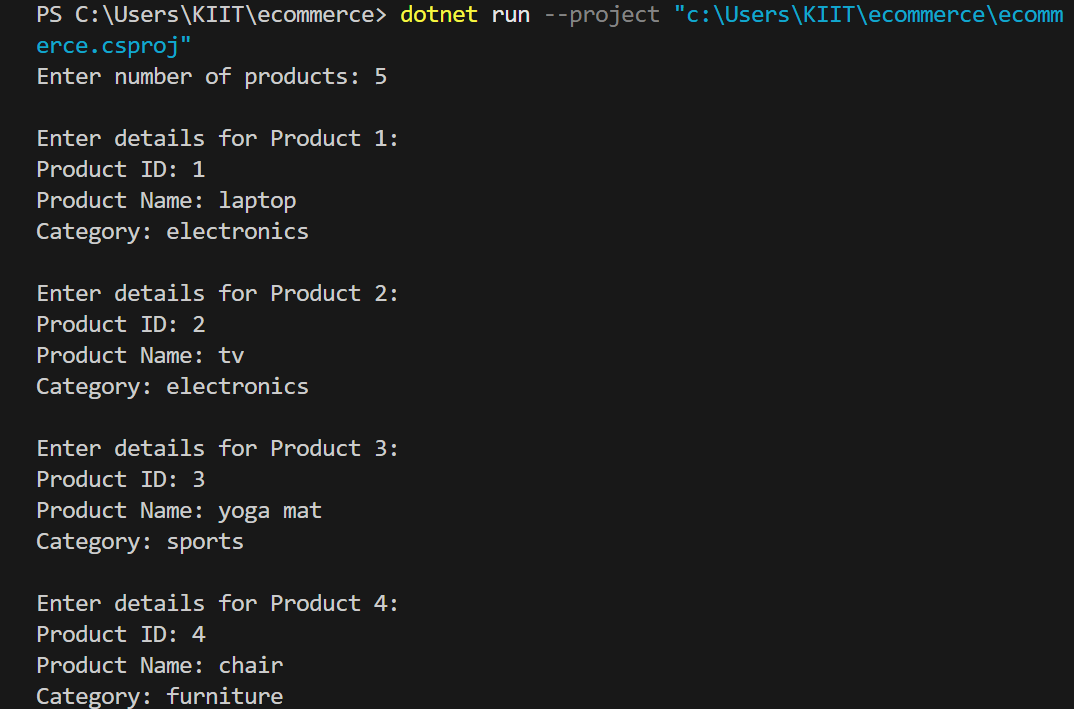
Console.WriteLine(resultBinary != null ? resultBinary.ToString() : "Product not found.");

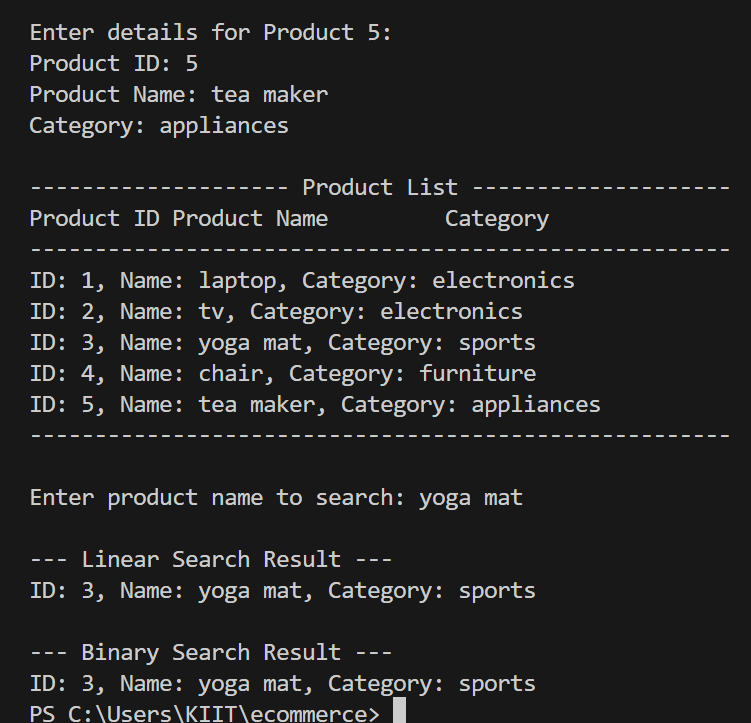
}

}

**OUTPUT:**

**After compiling, we are taking the user input(ProductID,Product and category) and showing them in the form of table and after that we are implementing the search algorithms by searching the product as given below in the screenshots.**

****

****

**EX.7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

Steps:

1. Understand Recursive Algorithms:
   * Explain the concept of recursion and how it can simplify certain problems.
2. Setup:
   * Create a method to calculate the future value using a recursive approach.
3. Implementation:
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. Analysis:
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

Solution:

1.Recursion is a programming concept where a function calls itself within its own definition to solve a problem. Here we use the STACK data structure.

It simplifies problems that:

* Have a repetitive, self-similar structure (e.g., calculating future values based on previous years).
* Can be broken down into base case + recursive case.

2. We'll define a recursive method to calculate the future value of an investment over time, based on:

* Initial value
* Annual growth rate
* Number of years to forecast.

3.**CODE:**

using System;

class FinancialForecast

{

    // Recursive method to calculate future value

    public static double CalculateFutureValue(double initialValue, double growthRate, int years)

    {

        // Base case: no more years to grow

        if (years == 0)

            return initialValue;

        // Recursive case: grow this year's value and call for remaining years

        return CalculateFutureValue(initialValue \* (1 + growthRate), growthRate, years - 1);

    }

    static void Main()

    {

        Console.Write("Enter initial value: ");

        double initial = double.Parse(Console.ReadLine());

        Console.Write("Enter annual growth rate (e.g., 0.05 for 5%): ");

        double rate = double.Parse(Console.ReadLine());

        Console.Write("Enter number of years: ");

        int years = int.Parse(Console.ReadLine());

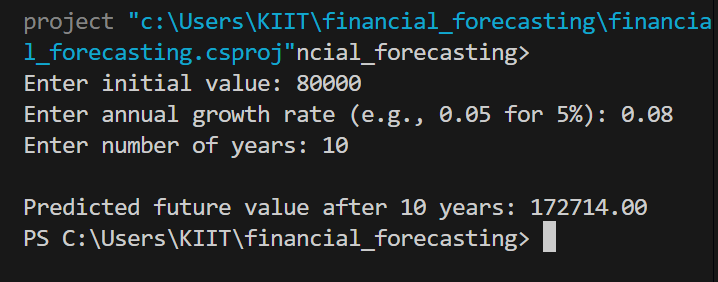
        double futureValue = CalculateFutureValue(initial, rate, years);

        Console.WriteLine($"\nPredicted future value after {years} years: {futureValue:F2}");

    }

}

**OUTPUT**:



**ANALYSIS:**

>**Time complexity**:

i)The algorithm runs O(n) times where n denotes the number of years.so it’s linear

ii)It makes one recursive call per year.

**Note:** For very large values of n, recursion can lead to **stack overflow.**

**Optimization:**

Use tail recursion or convert to iteration to avoid deep call stacks.

public static double CalculateFutureValueIterative(double initialValue, double growthRate, int years)

{

double result = initialValue;

for (int i = 0; i < years; i++)

{

result \*= (1 + growthRate);

}

return result;

}